



# 操作系统原理及应用

---

李 伟

xchlw@seu.edu.cn

计算机科学与工程学院、软件学院  
江苏省网络与信息安全重点实验室



# Chapter 4 Threads

---

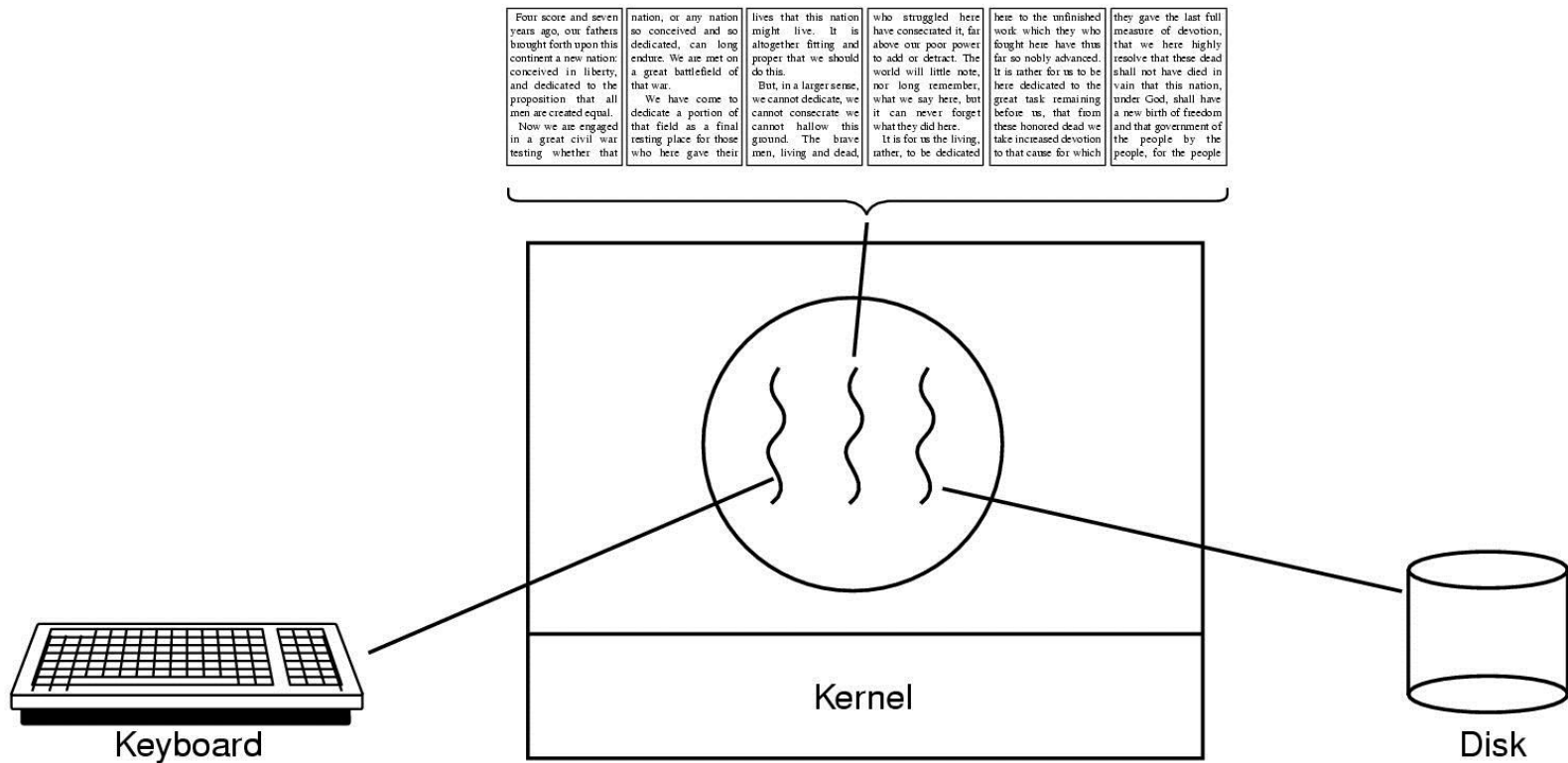


# Outline

---

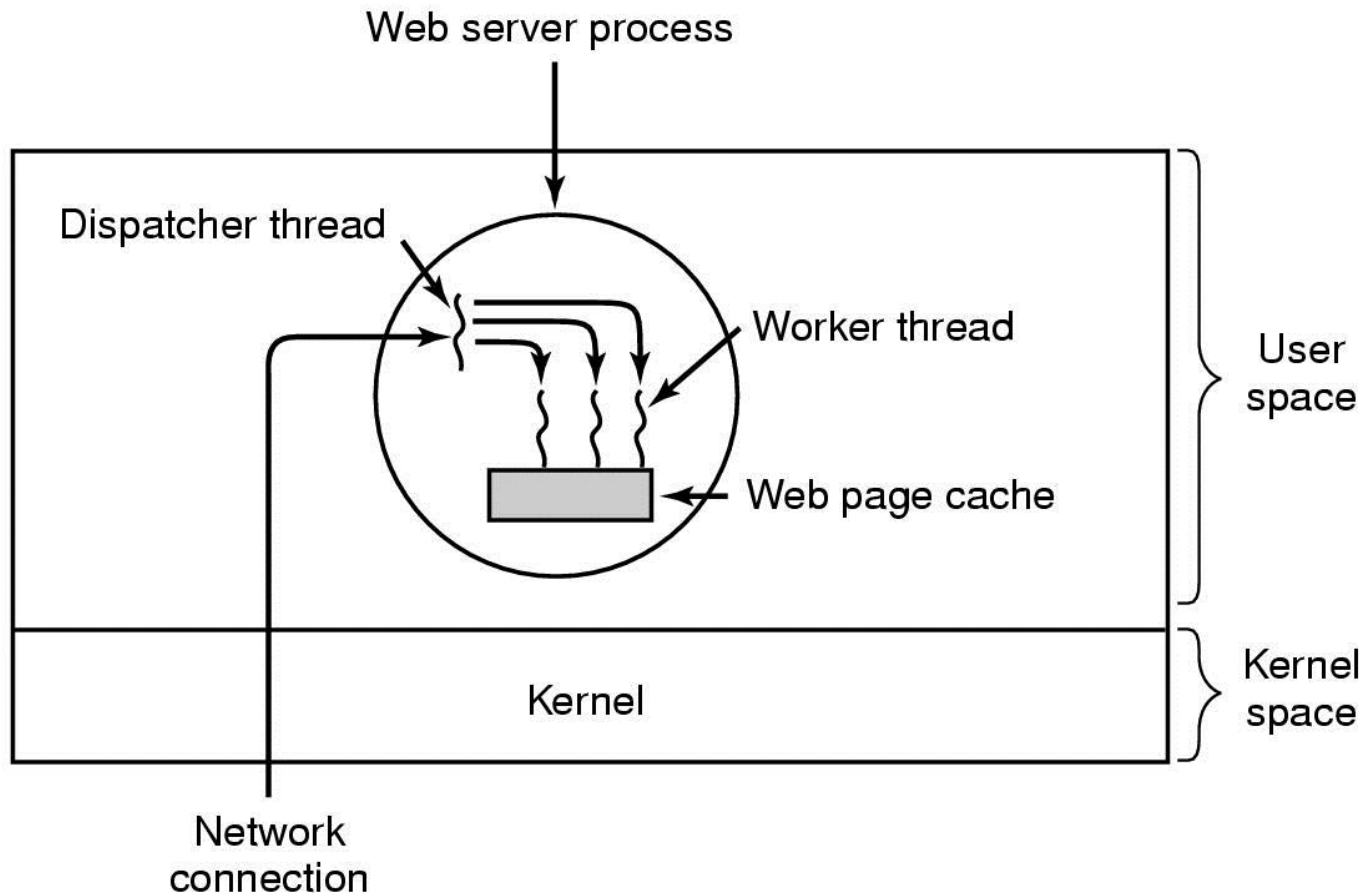
- **Overview**
- **Multithreading Models**
- **Thread Libraries**
- **Threading Issues**
- **Operating System Examples**

# Example (1/2)



**A word processor**

# Example (2/2)



**A Web server**



# What is a thread? (1/2)

---

- A thread is **a flow of control** within a process.
- A **thread**, also known as **LightWeight Process (LWP)**, is a **basic unit of CPU execution**.
- A traditional process, or heavyweight process, has **a single thread of control**.
- A multithreaded process contains several **different flows of control** within **the same address space**.

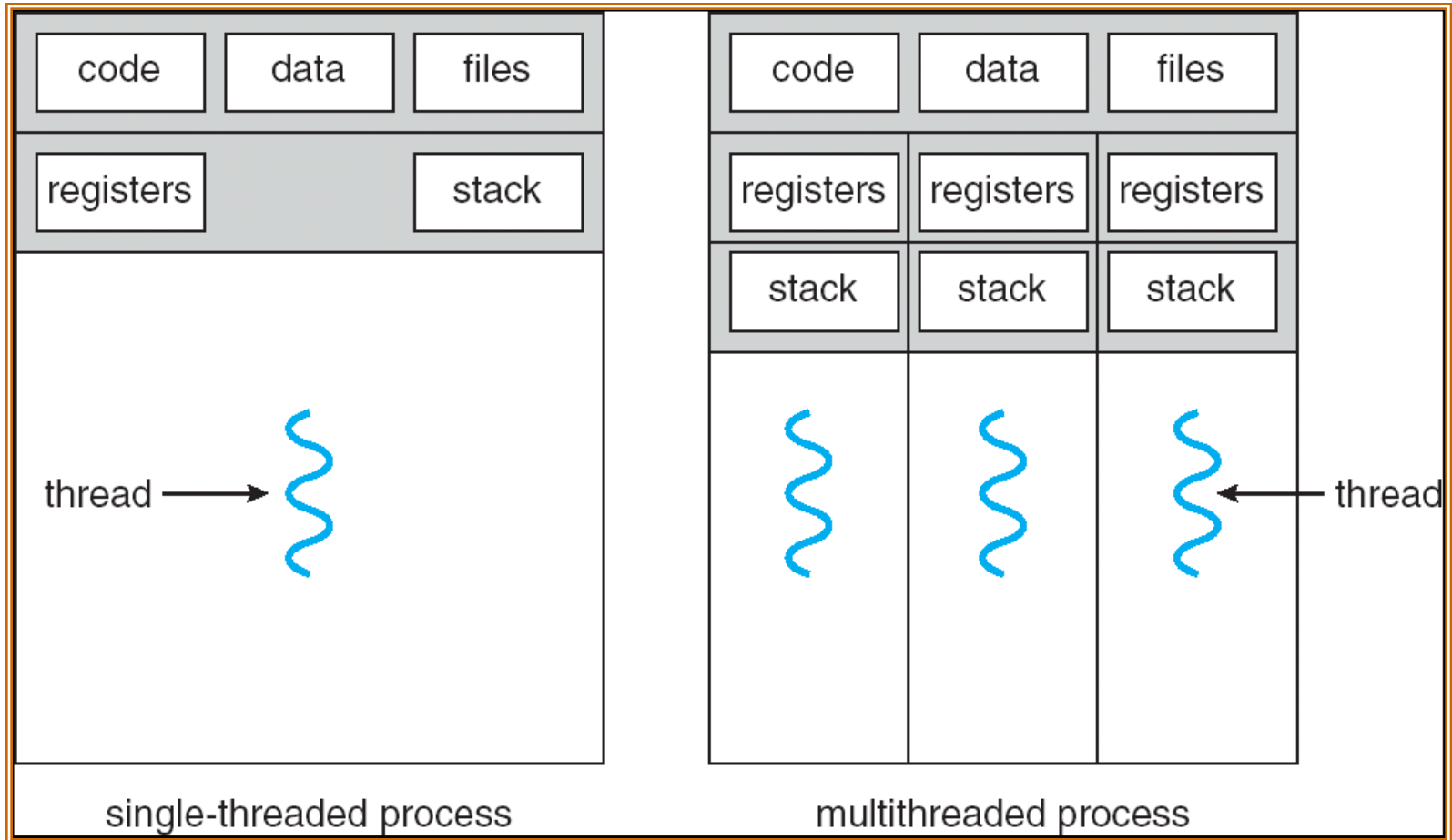


# What is a thread? (2/2)

---

- A thread has a **thread ID**, a **program counter**, a **register set**, and a **stack**. Thus, it is similar to a process has.
- However, a thread **shares** with other threads in the **same** process its **code section**, **data section**, and **other OS resources** (e.g., files and signals).

# Single and Multithreaded Processes







# Benefits

---

- **Responsiveness**
  - For instance, multithreaded web browser
- **Resource Sharing**
  - Threads share the memory and the resources of the process to which they belong
- **Economy**
  - The overhead of creating and context-switch threads is low.
- **Utilization of Multiprocessor Architectures**



# User Threads


---

- **Thread management done by user-level threads library**
- **Examples**
  - **POSIX Pthreads**
  - **Mach C-threads**
  - **Solaris UI-threads**

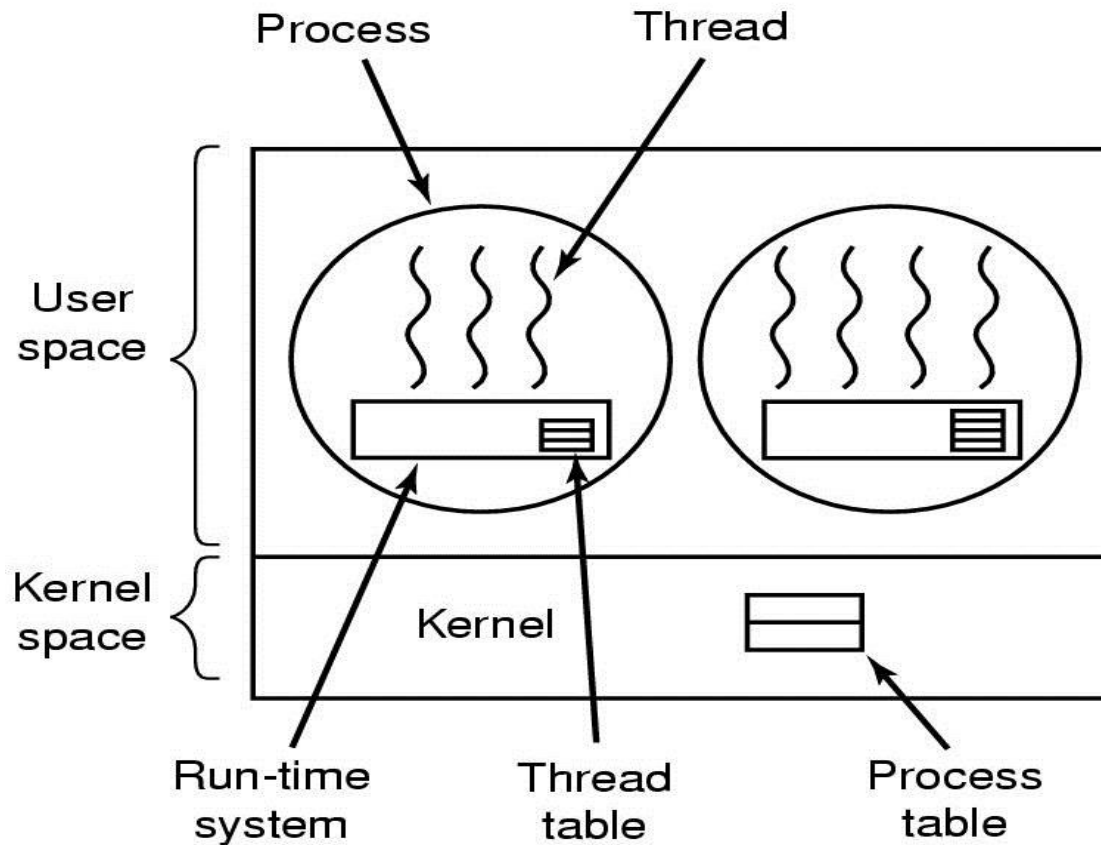


# User Threads

---

- User threads are supported above the kernel. The kernel **is not aware** of user threads.
- A library provides all support for thread creation, termination, joining, and scheduling.
- There is no kernel intervention, and, hence, user threads are usually **more efficient**. 高效率
- Unfortunately, since the kernel only recognizes the containing process (of the threads), ***if one thread is blocked, every other threads of the same process are also blocked*** because the containing process is blocked. 

# Implementing Threads in User Space



**A user-level threads package**



# Kernel Threads

---

- **Supported and managed by the Kernel**
- **Examples**
  - **Windows**
  - **Solaris**
  - **Mac OS X**
  - **Linux**

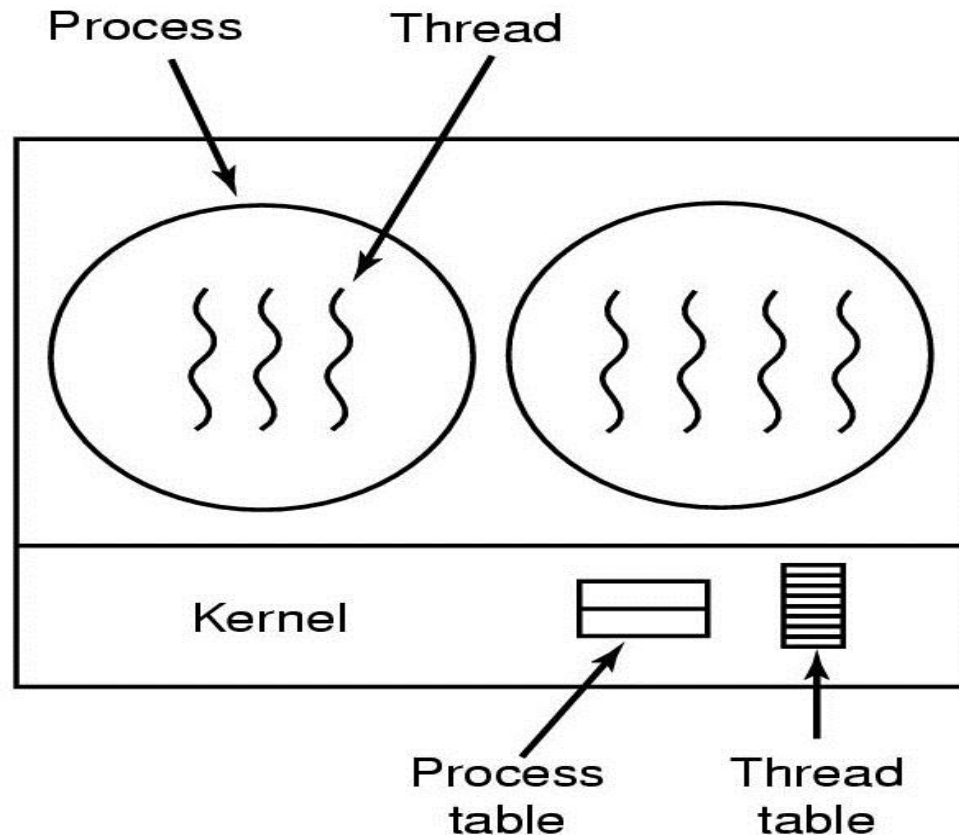


# Kernel Threads

---

- The kernel does thread creation, termination, joining, and scheduling in kernel space.
- Kernel threads are usually **slower** than the user threads.
- However, *blocking one thread will not cause other threads of the same process to block*. The kernel simply runs other threads.
- In a multiprocessor environment, the kernel can schedule threads on different processors

# Implementing Threads in the Kernel



**A threads package managed by the kernel**



# 作业1

---

1. 请阐述进程和线程之间的区别和联系。
2. 用户级线程和内核级线程有何区别？





# Outline

---

- Overview
- **Multithreading Models**
- Thread Libraries
- Threading Issues
- Operating System Examples



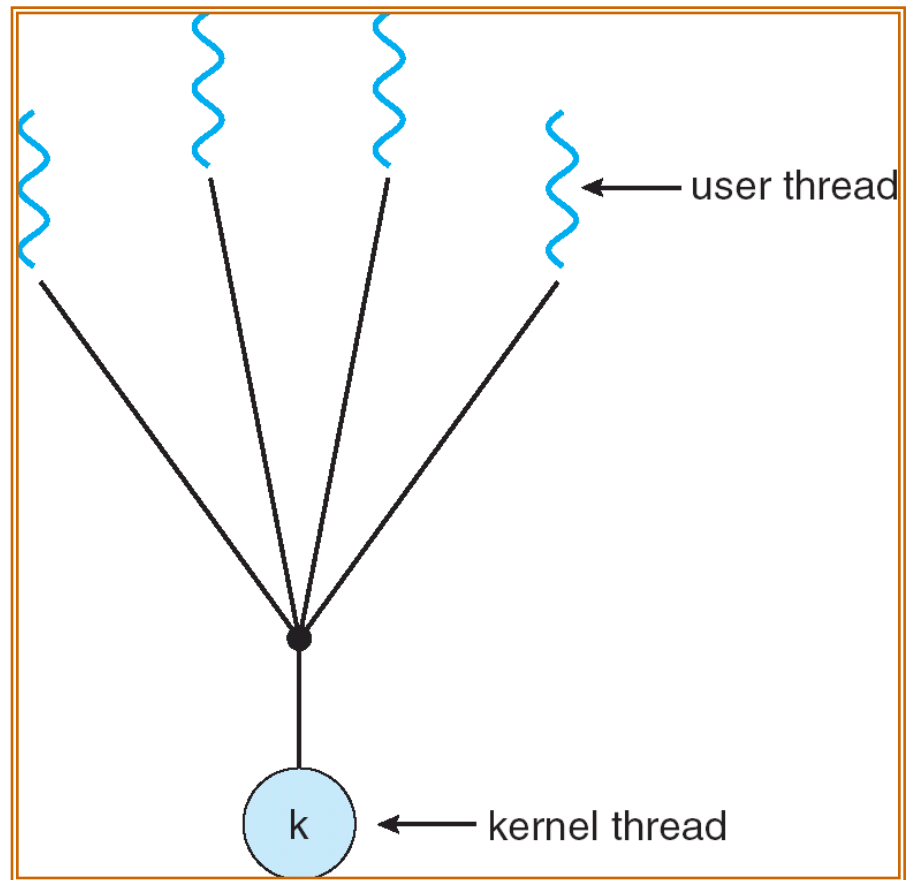
# Multithreading Models

---

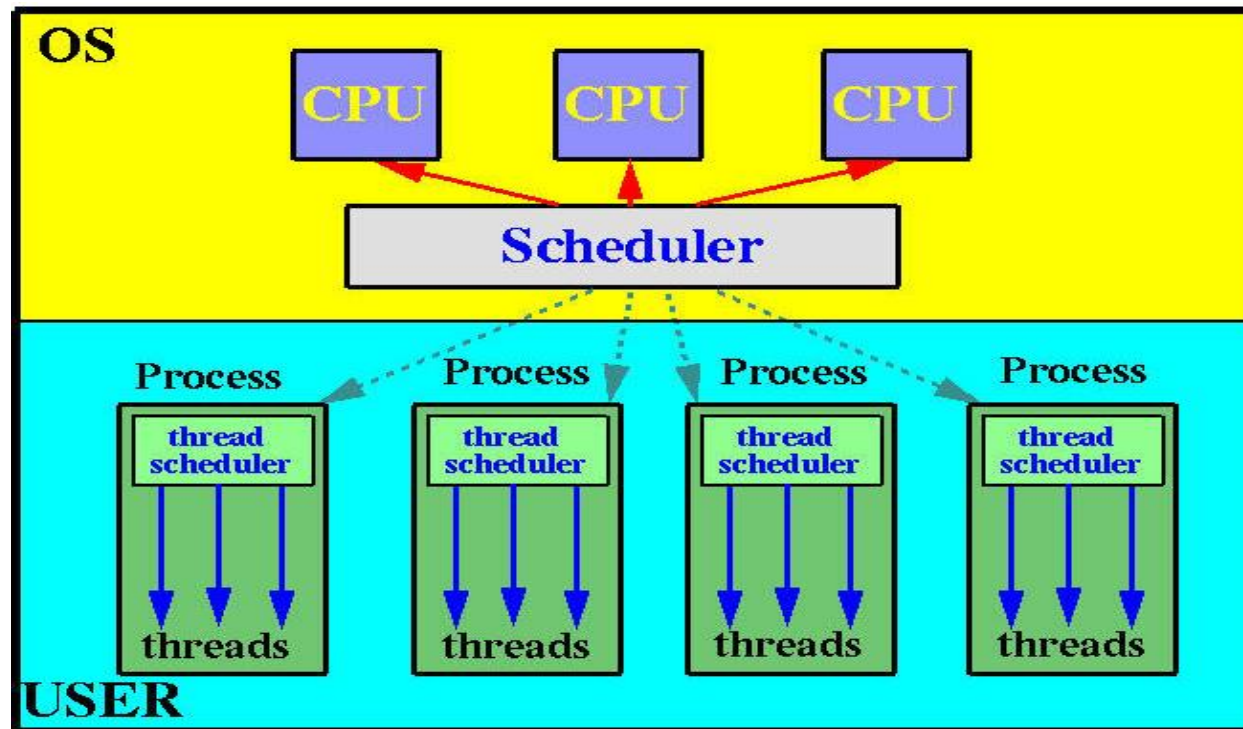
- **Many-to-One**
- **One-to-One**
- **Many-to-Many**

# Many-to-One Model

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.



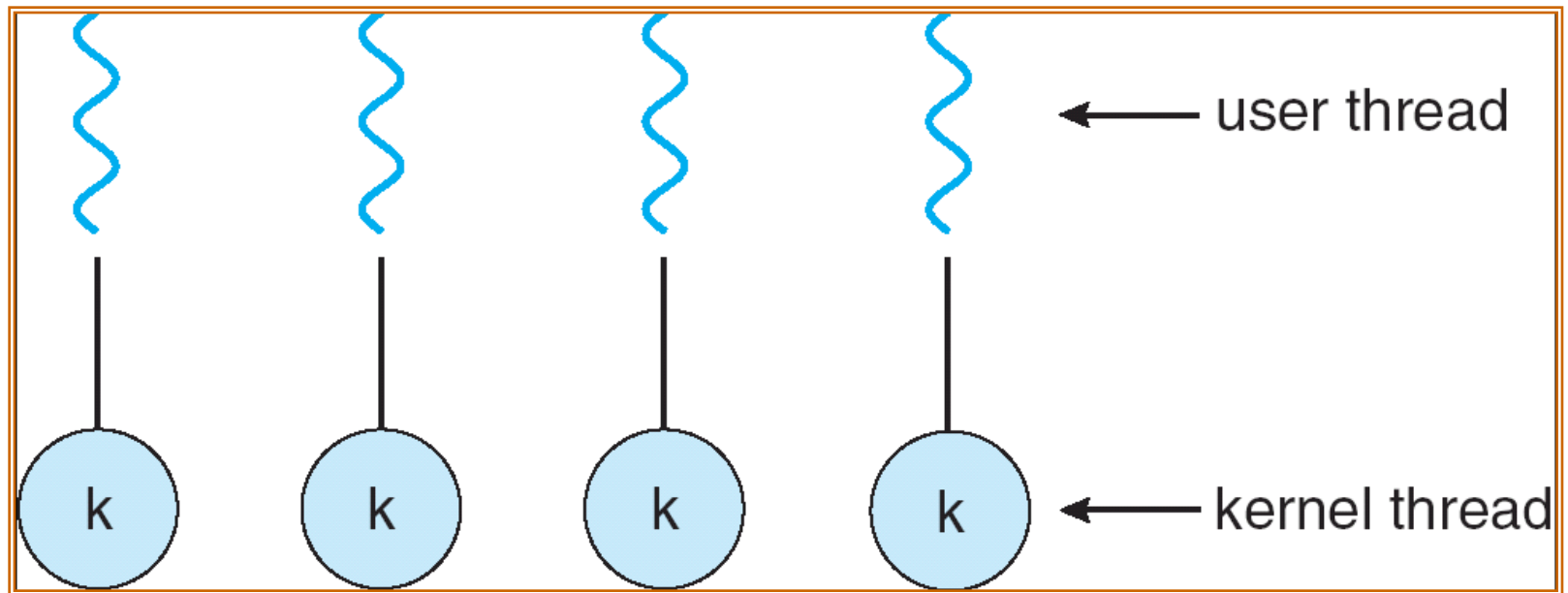
# Many-to-One Model



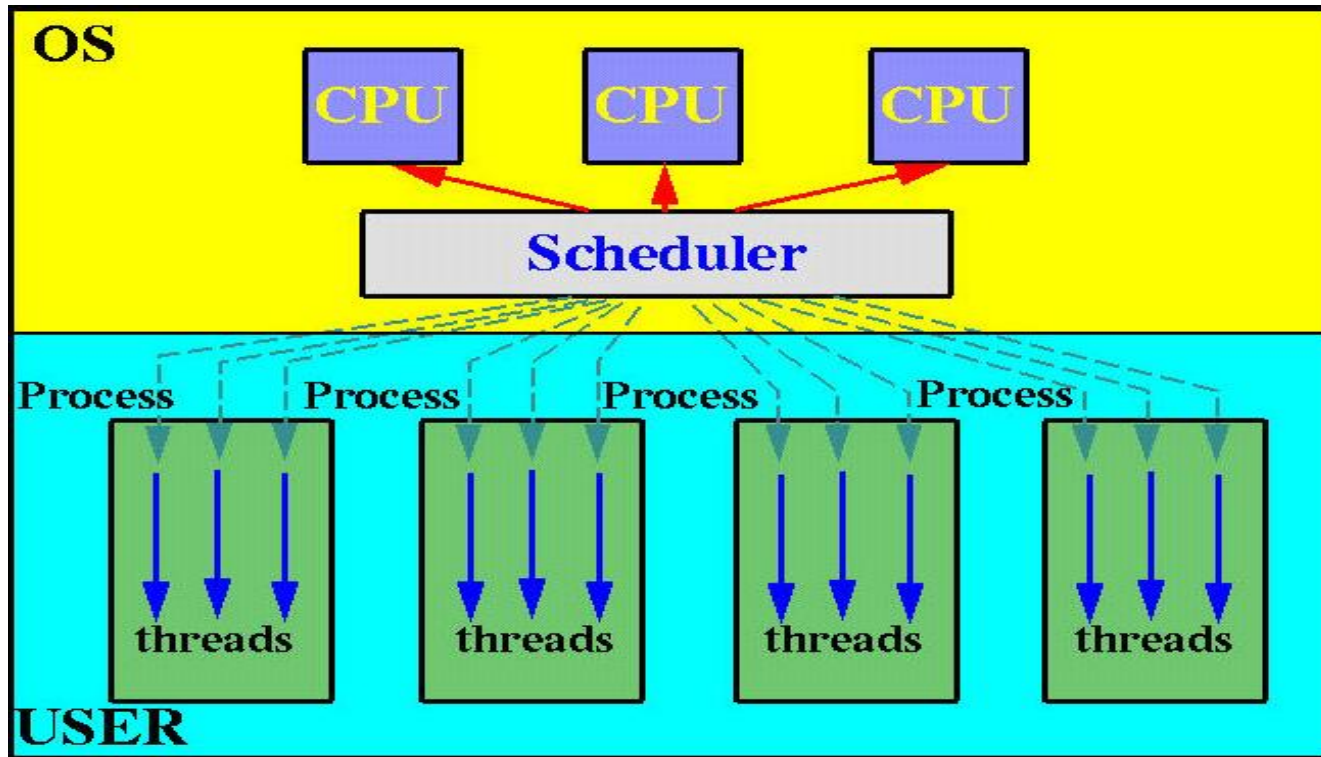
- Only one thread in the one process can access the kernel at a time.
- True concurrency is not gained.

# One-to-One Model

- Each user-level thread maps to kernel thread.
- Examples: Windows & Linux



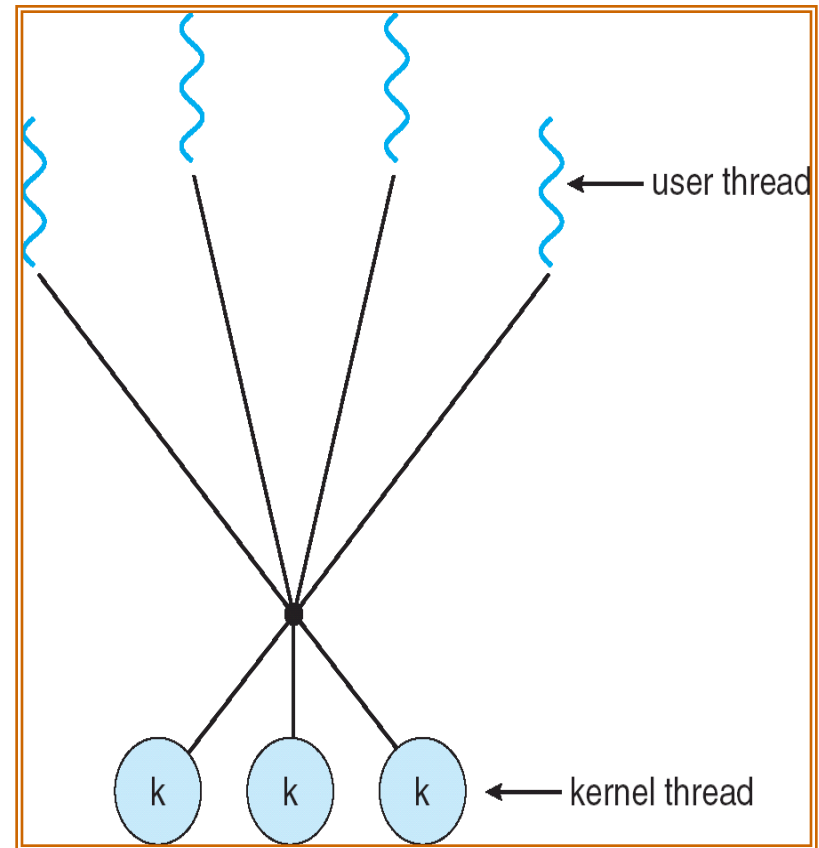
# One-to-one Model



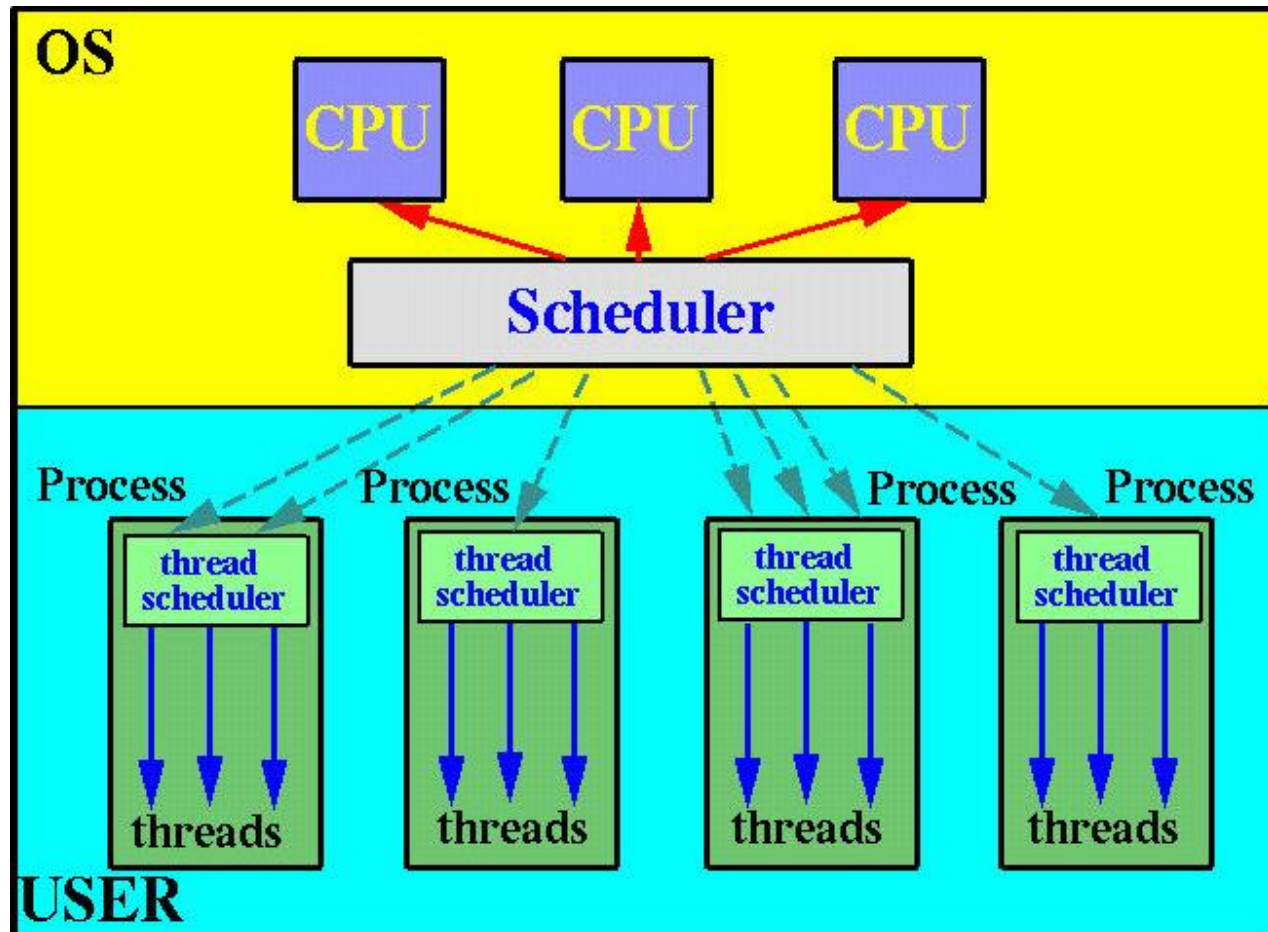
- Providing more concurrency.
- Restricting the number of threads supported by the system.

# Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Solaris 2
- Windows NT/2000 with the *ThreadFiber* package



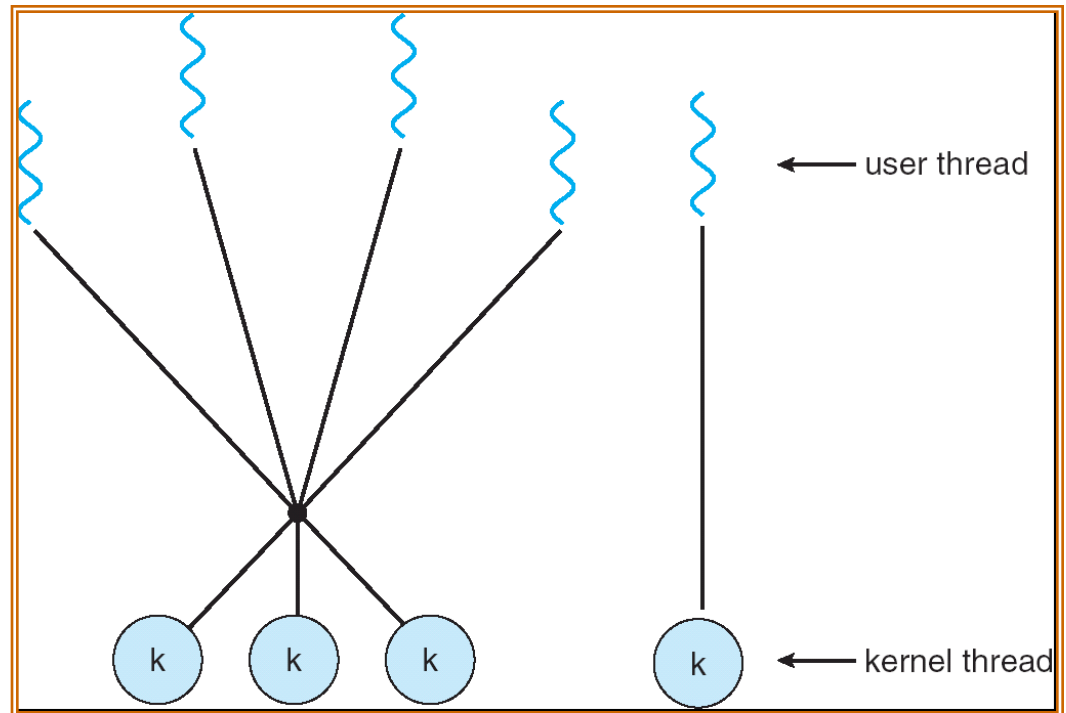
# Many-to-Many Model





# Two-level Model

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier





# Outline

---

- Overview
- Multithreading Models
- **Thread Libraries**
- Threading Issues
- Operating System Examples



# Thread Libraries

---

- A thread library provides the programmer an API for creating and managing threads.
  - POSIX Pthreads (User Level & Kernel Level)
  - Win32 (Kernel Level)
  - Java (Host OS)



# Pthreads

---

- **A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.**
- **API specifies behavior of the thread library, implementation is up to development of the library.**
- **Common in UNIX operating systems.**



# pthread\_create

```
int pthread_create(tid, attr, function, arg);
```

- **pthread\_t \*tid**
  - handle of created thread
- **const pthread\_attr\_t \*attr**
  - attributes of thread to be created
- **void \*(\*function) (void\*)**
  - function to be mapped to thread
- **void \*arg**
  - single argument to function



# pthread\_create explained

---

- spawn a thread running the function
- thread handle returned via pthread\_t structure
- specify **NULL** to use default attributes
- single argument sent to function
  - If no argument to function, specify **NULL**
- check error codes (returned value)!

**EAGAIN** – insufficient resources to create thread

**EINVAL** – invalid attribute



# Threads states

---

- **pthread threads have two states**
  - joinable and detached
- **threads are joinable by default**
  - Resources are kept until *pthread\_join ( )*
  - can be reset with attribute or API call
- **detached thread can not be joined**
  - resources can be reclaimed at termination
  - cannot reset to be *joinable*



# Waiting for a thread

---

```
int pthread_join(tid, val_ptr);
```

- **pthread\_t \*tid**
  - **handle of joinable thread**
- **void \*\*val\_ptr**
  - **exit value returned by joined thread**





# **pthread\_join explained**

---

- **calling thread waits for the thread with handle tid to terminate**
- **only one thread can be joined**
- **thread must be joinable**
  - **exit value is returned from joined thread**
  - **type returned is (void \*)**
  - **use NULL if no return value expected**

**ESRCH –thread not found**

**EINVAL – thread not joinable**



# Example: Multiple threads

---

```
#include <stdio.h>
#include <pthread.h>
const int numThreads = 4;

void *helloFunc(void * pArg)
{   printf("Hello Thread\n"); }

main()
{   pthread_t hThread[numThreads];
    for (int i = 0; i < numThreads; i++)
        pthread_create(&hThread[i], NULL, helloFunc, NULL);
    for (int i = 0; i < numThreads; i++)
        pthread_join(hThread[i], NULL);
    return 0;
}
```



# Thread Termination

---

- **void pthread\_exit(void \*status);**
  - terminate the current thread
- **int pthread\_cancel(pthread\_t thread);**
  - the thread may:
    - ignore the request
    - terminated immediately (Asynchronous cancellation)
    - deferred terminated (Deferred cancellation)
- **int pthread\_kill(pthread\_t thread, int sig);**



# Thread Cancellation

---

- **int pthread\_setcancelstate(int state, int \*oldstate);**
  - **PTHREAD\_CANCEL\_ENABLE**
  - **PTHREAD\_CANCEL\_DISABLE**
- **int pthread\_setcanceltype(int type, int \*oldtype);**
  - **PTHREAD\_CANCEL\_ASYNCHROUS**
  - **PTHREAD\_CANCEL\_DEFERRED**
- **void pthread\_testcancel(void);**



# Windows Thread APIs

---

- **CreateThread**
- **GetCurrentThreadId** - returns global ID
- **GetCurrentThread** - returns handle
- **SuspendThread/ResumeThread**
- **ExitThread**
- **TerminateThread**
- **GetExitCodeThread**
- **GetThreadTimes**



# Windows API Thread Creation

```
HANDLE CreateThread (  
    LPSECURITY_ATTRIBUTES lpsa,  
    DWORD cbStack,  
    LPTHREAD_START_ROUTINE lpStartAddr,  
    LPVOID lpvThreadParm,  
    DWORD fdwCreate,  
    LPDWORD lpIDThread)
```

cbStack == 0: thread's  
stack size defaults to  
primary thread's size

lpStartAddr points to function declared as

```
DWORD WINAPI ThreadFunc(LPVOID)
```

- lpvThreadParm is 32-bit argument
- lpIDThread points to DWORD that receives thread ID  
non-NULL pointer !



# Windows API Thread Termination

---

**VOID ExitThread( DWORD devExitCode )**

- When the last thread in a process terminates, the process itself terminates

**BOOL GetExitCodeThread (**  
**HANDLE hThread, LPDWORD lpdwExitCode)**

- Returns exit code or STILL\_ACTIVE



# Suspending and Resuming Threads

---

- Each thread has suspend count
- Can only execute if suspend count == 0
- Thread can be created in suspended state

```
DWORD ResumeThread (HANDLE hThread)  
DWORD SuspendThread(HANDLE hThread)
```

- Both functions return suspend count or 0xFFFFFFFF on failure





# Example: Thread Creation

```
#include <stdio.h>
#include <windows.h>

DWORD WINAPI helloFunc(LPVOID arg ) {
    printf("Hello Thread\n");
    return 0;
}

main() {
    HANDLE hThread =
        CreateThread(NULL, 0, helloFunc,
                    NULL, 0, NULL );
}
```



What's Wrong?



# Example Explained

---

- **Main thread is process**
- **When process goes, all threads go**
- **Need some methods of waiting for a thread to finish**

# Waiting for Windows Thread

```
#include <stdio.h>
#include <windows.h>
BOOL thrdDone = FALSE;

DWORD WINAPI helloFunc(LPVOID arg ) {
    printf("Hello Thread\n");
    return 0;
}

main() {
    HANDLE hTh
        CreateThread(NULL, 0, helloFunc,
                     NULL, 0, NULL );
}
```

**thrdDone = TRUE;**

**Not a good idea!**

**while (!thrdDone);**



# Waiting for a Thread

Wait for one object (thread)

```
DWORD WaitForSingleObject(  
    HANDLE hHandle,  
    DWORD dwMilliseconds );
```

Calling thread waits (blocks) until

- Time expires
  - Return code used to indicate this
- Thread exits (handle is signaled)
  - Use INFINITE to wait until thread termination

Does not use CPU cycles



# Waiting for Many Threads

Wait for up to 64 objects (threads)

```
DWORD WaitForMultipleObjects(  
    DWORD nCount,  
    CONST HANDLE *lpHandles, // array  
    BOOL fWaitAll, // wait for one or all  
    DWORD dwMilliseconds)
```

Wait for all: `fWaitAll==TRUE`

Wait for any: `fWaitAll==FALSE`

- Return value is first array index found



# Notes on WaitFor Functions

---

- **Handle as parameter**
- **Used for different types of objects**
- **Kernel objects have two states**
  - **Signaled**
  - **Non-signaled**
- **Behavior is defined by object referred to by handle**
  - **Thread: signaled means terminated**



# Example: Waiting for multiple threads

```
#include <stdio.h>
#include <windows.h>
const int numThreads = 4;

DWORD WINAPI helloFunc(LPVOID arg ) {
    printf("Hello Thread\n");
    return 0; }

main() {
    HANDLE hThread[numThreads];
    for (int i = 0; i < numThreads; i++)
        hThread[i] =
            CreateThread(NULL, 0, helloFunc, NULL, 0, NULL );
    WaitForMultipleObjects(numThreads, hThread,
                           TRUE, INFINITE);
}
```



# Example: HelloThreads

---

- **Modify the previous example code to print out**
  - **appropriate “Hello Thread” message**
  - **Unique thread number**
    - **use for-loop variable of CreateThread loop**





## Discussion: *What's Wrong?*

```
DWORD WINAPI threadFunc(LPVOID pArg) {
    int* p = (int*)pArg;
    int myNum = *p;
    printf( "Thread number %d\n", myNum);
}

. . .
// from main():
for (int i = 0; i < numThreads; i++) {
    hThread[i] =
        CreateThread(NULL, 0, threadFunc, &i, 0, NULL);
}
}
```

**What is printed for myNum?**



# Hello Threads Timeline

<i>Time</i>	<i>main</i>	<i>Thread 0</i>	<i>Thread 1</i>
T <sub>0</sub>	i = 0	---	----
T <sub>1</sub>	create(&i)	---	---
T <sub>2</sub>	i++ (i == 1)	launch	---
T <sub>3</sub>	create(&i)	p = pArg	---
T <sub>4</sub>	i++ (i == 2)	myNum = *p myNum = 2	launch
T <sub>5</sub>	wait	print(2)	p = pArg
T <sub>6</sub>	wait	exit	myNum = *p myNum = 2



# Race Conditions

---

- **Concurrent access of same variable by multiple threads**
  - **Read/Write conflict**
  - **Write/Write conflict**
- **Most common error in concurrent programs**
- **May not be apparent at all times**
- **How to avoid data races?**
  - **Local storage**
  - **Control shared access with **critical regions****



# Hello Thread: Local Storage solution

---

```
DWORD WINAPI threadFunc(LPVOID pArg)
{
    int myNum = *((int*)pArg);
    printf( "Thread number %d\n", myNum);
}

. . .

// from main():
for (int i = 0; i < numThreads; i++) {
    tNum[i] = i;
    hThread[i] =
        CreateThread(NULL, 0, threadFunc, &tNum[i],
                    0, NULL);
}
```

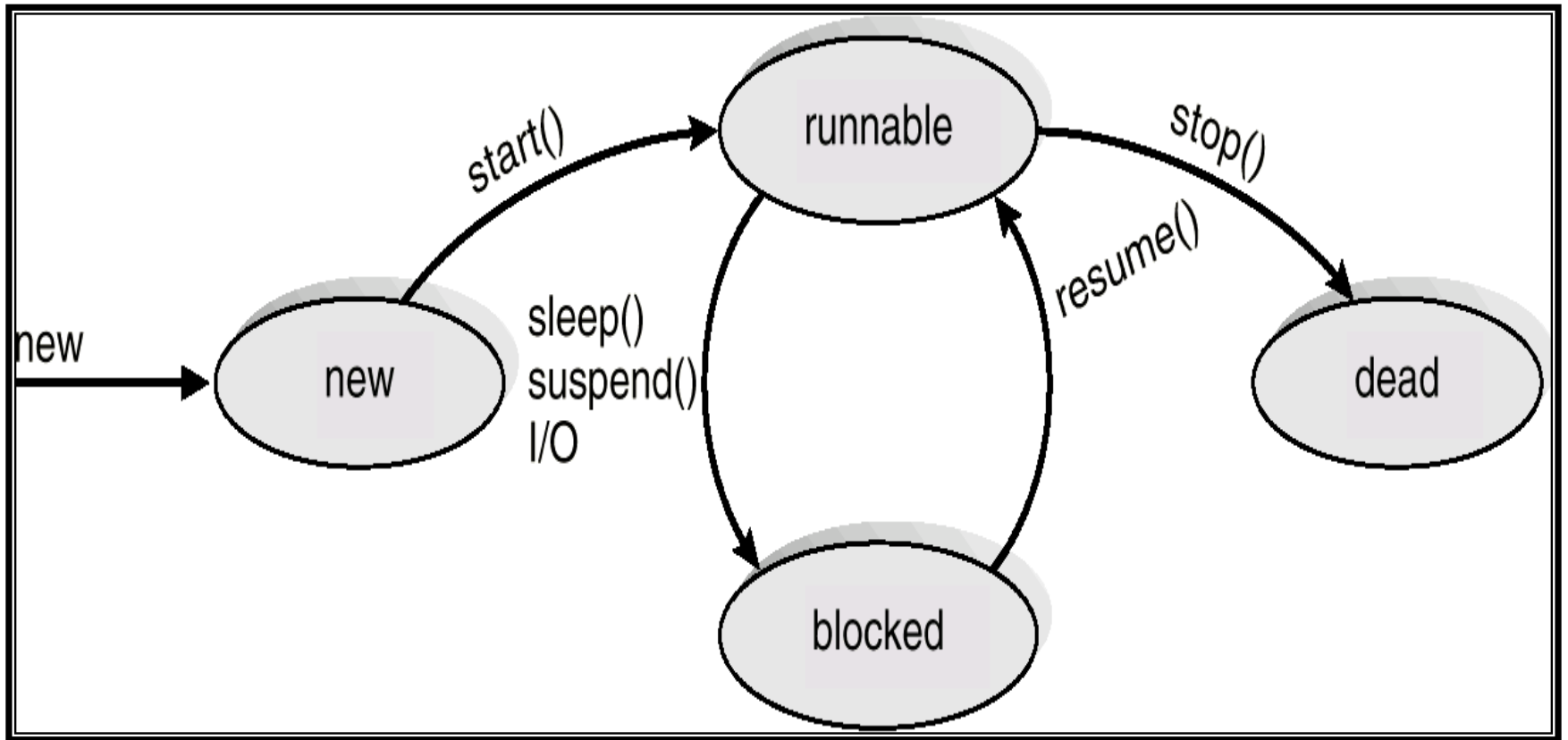


# Java Threads

---

- **Java threads may be created by**
  - **Extending Thread class to create a new class**
  - **Defining a class that Implements the Runnable interface**
- **Java threads are managed by the JVM.**

# Java Thread States





# Outline

---

- Overview
- Multithreading Models
- Thread Libraries
- **Threading Issues**
- Operating System Examples



# Threading Issues

---

- **Semantics of fork() and exec() system calls**
- **Thread cancellation**
- **Signal handling**
- **Thread pools**
- **Thread specific data**
- **Scheduler activations**





# Semantics of fork() and exec()

---

- Does fork() duplicate only the calling thread or all threads? ***ALL OK!***
- If a thread invokes the exec(), what will be the result?

**The program specified in the parameter to exec() will replace the entire process—including all threads.**



# Thread Cancellation

---

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - The point a thread can terminate itself is a *cancellation point*
    - `void pthread_testcancel(void);`



# Thread Cancellation

---

- With **asynchronous cancellation**, if the target thread owns some system-wide resources, the system may not be able to reclaim all resources.
- With **deferred cancellation**, the target thread determines the time to terminate itself. Reclaiming resources is not a problem.
- Most systems implement **asynchronous cancellation** for processes (e.g., use the kill system call) and threads.
- **Pthread** supports **deferred cancellation**.



# Signal Handling

---

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- All signals follow the same pattern:
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled
- ***A signal handler*** is used to process signals



# Signal Handling

---

- **Where should a signal be delivered in multithreaded programs?**
  - **Deliver the signal to the thread to which the signal applies**
  - **Deliver the signal to every thread in the process**
  - **Deliver the signal to certain threads in the process**
  - **Assign a specific thread to receive all signals for the process**



## 作业2

---

- 试比较信号机制与中断机制的异同。



# Thread Pools

---

- **Create a number of threads in a pool where they await work**
- **Advantages:**
  - **Usually slightly faster to service a request with an existing thread than create a new thread**
  - **Allows the number of threads in the application(s) to be bound to the size of the pool**



# Thread Specific Data

---

- Threads belonging to a process share the data of the process.
- Allows each thread to have its **own copy of data**
- when using a thread pool, each thread may be assigned a unique identifier





# Scheduler Activations

---

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number of kernel threads

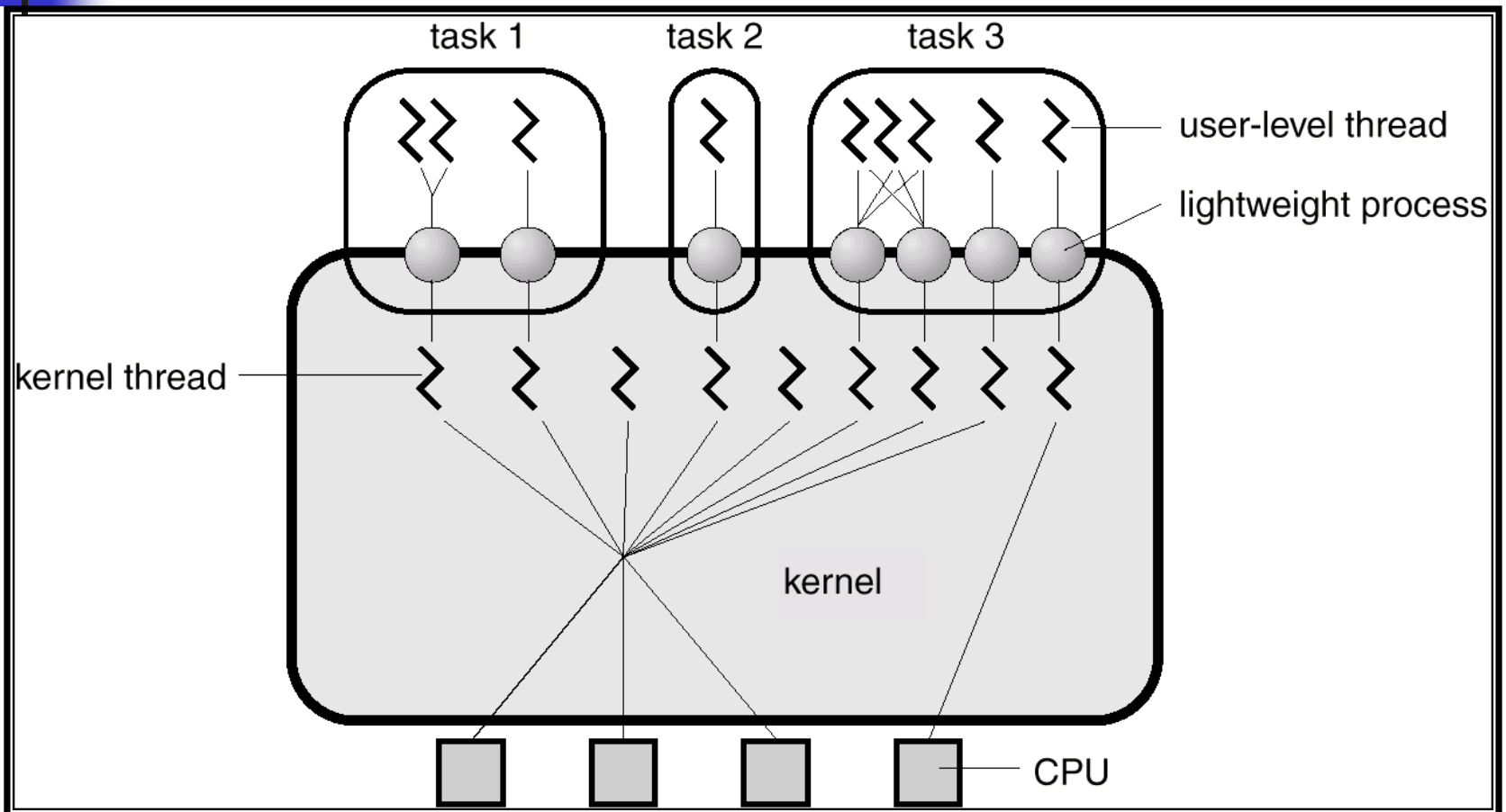


# Outline

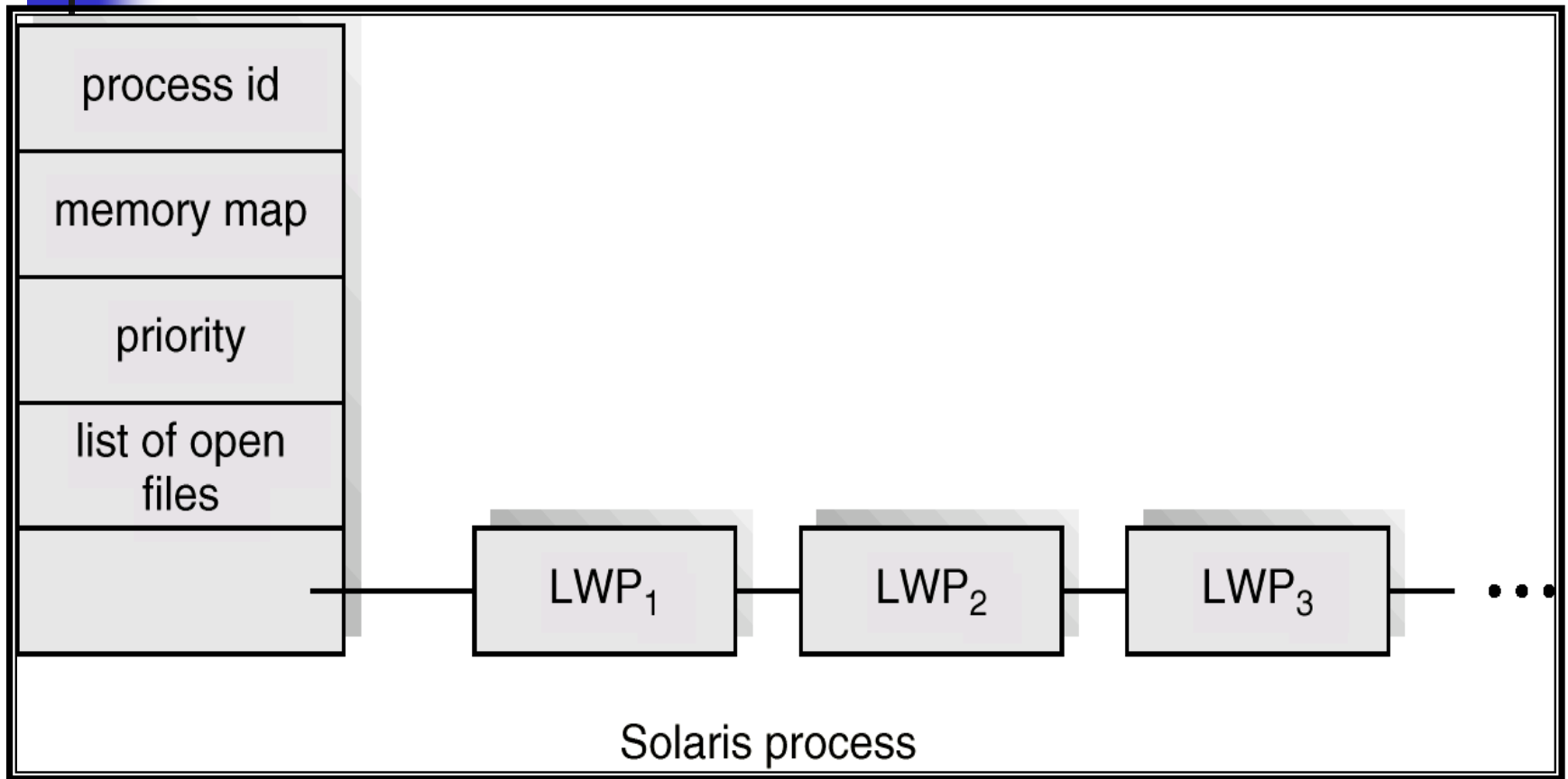
---

- **Overview**
- **Multithreading Models**
- **Thread Libraries**
- **Threading Issues**
- **Operating System Examples**

# Solaris 2 Threads



# Solaris Process





# Windows XP Threads

---

- **Implements the one-to-one mapping.**
- **Each thread contains**
  - **a thread id**
  - **register set**
  - **separate user and kernel stacks**
  - **private data storage area**



# Linux Threads

---

- Linux does not distinguish between processes and threads.
- Linux refers to them as *tasks* rather than *threads*.
- Thread creation is done through **clone() system call**.
- Clone() allows a child task to share the address space of the parent task (process)